3-2016

# RACK: Automatic API recommendation using crowdsourced knowledge

Mohammad M. RAHMAN

Chanchal K. ROY

David LO

*Singapore Management University*, davidlo@smu.edu.sg

**DOI:** https://doi.org/10.1109/SANER.2016.80

2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering

# RACK: Automatic API Recommendation using Crowdsourced Knowledge

Mohammad Masudur Rahman Chanchal K. Roy †David Lo
University of Saskatchewan, Canada, †Singapore Management University, Singapore
{masud.rahman, chanchal.roy}@usask.ca, †davidlo@smu.edu.sg

***Abstract*—Traditional code search engines often do not perform well with natural language queries since they mostly apply keyword matching. These engines thus need carefully designed queries containing information about programming APIs for code search. Unfortunately, existing studies suggest that preparing an effective code search query is both challenging and time consuming for the developers. In this paper, we propose a novel API recommendation technique–RACK that recommends a list of relevant APIs for a natural language query for code search by exploiting keyword-API associations from the crowdsourced knowledge of Stack Overflow. We first motivate our technique using an exploratory study with 11 core Java packages and 344K Java posts from Stack Overflow. Experiments using 150 code search queries randomly chosen from three Java tutorial sites show that our technique recommends correct API classes within the top 10 results for about 79% of the queries which is highly promising. Comparison with two variants of the state-of-the-art technique also shows that RACK outperforms both of them not only in Top-K accuracy but also in mean average precision and mean recall by a large margin.**

***Index Terms*—Code search, query reformulation, keyword-API association, crowdsourced knowledge, Stack Overflow**

## I. Introduction

Studies show that software developers on average spend about 19% of their development time in web search where they mostly look for relevant code snippets for their tasks [13]. Code search engines such as Open Hub, Koders, GitHub search and Krugle provide access to thousands of large open source projects which are potential sources for such snippets [21]. Traditional code search engines generally employ keyword matching, i.e., return code snippets based on lexical similarity between search query and source code. They expect carefully designed queries containing relevant API classes or methods from the users, and thus, often do not perform well with unstructured natural language queries. Unfortunately, preparing an effective search query containing information about relevant APIs is not only a challenging but also a time-consuming task for the developers [13, 19]. Previous study also suggested that on average, developers with varying experience levels performed poorly in coming up with good search terms for code search [19]. Thus, an automated technique that translates a natural language query into a set of relevant API classes or methods (i.e., search-engine friendly query) can greatly assist the developers in code search. Our paper addresses this particular research problem by exploiting the crowdsourced knowledge from Stack Overflow Q & A site.

Existing studies on API recommendation accept one or more natural language queries, and return relevant API classes and methods by analyzing feature request history and API documentations [29], API invocation graphs [14], library usage patterns [28], code surfing behaviour of the developers and API invocation chains [21]. McMillan et al. [21] first propose *Portfolio* that recommends relevant API methods for a given code search query, and demonstrates their usage from a large codebase. Chan et al. [14] improve upon *Portfolio* by employing further sophisticated graph-mining and textual similarity techniques. Thung et al. [29] recommend relevant API methods to assist the implementation of an incoming feature request. Although all these techniques perform well in different working contexts, they share a set of limitations and fall short to address our research problem. First, each of these techniques [14, 21, 29] exploits lexical similarity measure (e.g., Dice's coefficients [14]) for candidate API selection. This warrants that the search query should be carefully prepared, and it should contain keywords similar to the API names. In other words, the developer should possess a certain level of experience on the target APIs to actually use those techniques [12]. Second, API names and search queries are generally provided by different developers who may use different vocabularies to convey the same concept [20]. Concept location community has termed it as vocabulary mismatch problem [17]. Lexical similarity based techniques often suffer from this problem. Hence, the performance of these techniques is not only limited but also subject to the identifier naming practices adopted in the codebase under study. We thus need a technique that overcomes the above limitations, and recommends relevant APIs for natural language queries from a wider vocabulary.

One possible way to tackle the above challenges is to exploit crowdsourced knowledge on the usage of particular API classes and methods. Let us consider a natural language query–*"Generating MD5 hash of a Java string."* Now, we analyze thousands of Q & A posts from Stack Overflow that suggest relevant APIs for this task, and then recommend APIs from them. For instance, the Q & A example in Fig. 1 discusses on how to generate an MD5 hash (Fig. 1-(a)), and the accepted answer (Fig. 1-(b)) suggests that `MessageDigest` API should be used for the task. Such usage of the API is also recommended by at least 305 technical users from Stack Overflow which validates the appropriateness of the usage. Our work is thus generic, language independent, project insensitive, and in the same time, it overcomes the vocabulary mismatch problem suffered from by the past studies.

In this paper, we propose an API recommendation

978-1-5090-1855-0/16 $31.00 © 2016 IEEE
DOI 10.1109/SANER.2016.80

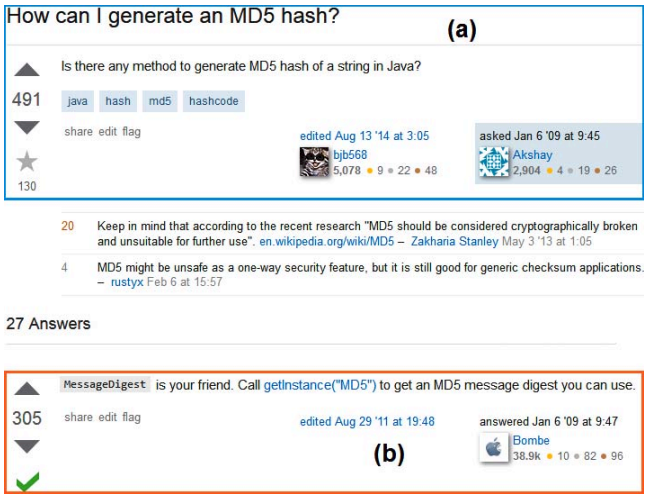


Fig. 1. An example of Stack Overflow (a) question & (b) accepted answer

technique–RACK–that exploits the association of different APIs with query keywords from Stack Overflow, and translates a natural language query for code search into a set of relevant APIs. First, we motivate our idea of using crowdsourced knowledge for API recommendation with an exploratory study where we analyze 172,043 questions and their accepted answers from Stack Overflow. Second, we construct a keyword-API mapping database using those questions and answers where the keywords (i.e., programming requirements) are extracted from questions and the APIs (i.e., programming solutions) are collected from corresponding accepted answers. Third, we propose an API recommendation technique that employs two heuristics on keyword-API associations, and recommends a ranked list of API classes for a given query. The baseline idea is to capture and learn the responses from millions of technical users (e.g., developers, researchers, programming hobbyists) for different programming problems, and then exploit them for relevant API recommendation. Our technique (1) does not rely on the lexical similarity between query and source code of projects for API selection, and (2) addresses the vocabulary mismatch problem by using a significantly large vocabulary (i.e., 20K) produced by millions of users of Stack Overflow. Thus, it has a great potential to overcome the challenges faced with the past studies.

An exploratory study with 344,086 Java related posts from Stack Overflow shows that (1) each post uses at least two different API classes on average, and (2) about 65% of the classes from each of the 11 core Java API packages are used in those posts. This suggests the potential of Stack Overflow for relevant API recommendation. Experiments using 150 code search queries randomly chosen from three Java tutorial sites show that our technique can recommend relevant APIs with a Top-10 accuracy of about 79% which is highly promising. We also compared with two variants of the state-of-the-art technique by Thung et al. [29], and report that our technique outperformed both of them not only in Top-K accuracy but also in mean average precision and mean recall by a large margin. Thus, the paper makes the following contributions:

TABLE I
API PACKAGES FOR EXPLORATORY STUDY

| Package | #Class | Package | #Class |
|---|---|---|---|
| `java.lang` | 255 | `java.net` | 84 |
| `java.util` | 470 | `java.security` | 148 |
| `java.io` | 105 | `java.awt` | 423 |
| `java.math` | 09 | `java.sql` | 29 |
| `java.nio` | 189 | `java.swing` | 1,195 |
| `java.applet` | 05 | | |

- An exploratory study that suggests the potential of Stack Overflow for relevant API recommendation for code search using natural language queries.
- A keyword-API mapping database that maps 655K question keywords to 551K API classes from Stack Overflow.
- A novel technique that exploits query keyword-API associations from crowdsourced knowledge, and translates a natural language query into a set of relevant API classes.
- Comprehensive evaluation of the proposed technique with four metrics, and comparison with the state-of-the-art.

## II. EXPLORATORY STUDY

Our technique relies on the mapping between natural language keywords from the questions of Stack Overflow and API classes from corresponding accepted answers for translating a code search query into relevant API classes. Thus, an investigation is warranted if such answers contain any API related information and the questions contain any query keywords. We perform an exploratory study using 344K posts from Stack Overflow, and analyze the usage and coverage of standard Java API classes in those posts. We also explore if the question titles are a potential source for keywords for code search. We particularly answer three research questions as follows:

- **RQ$_1$**: To what extent do the accepted answers from Stack Overflow refer to standard Java API classes?
- **RQ$_2$**: To what extent are the API classes from each of the core Java packages covered (i.e., mentioned) in the accepted answers from Stack Overflow?
- **RQ$_3$**: Do the titles from Stack Overflow questions contain potential keywords for code search?

### A. Data Collection

We collect 172,043 questions and their accepted answers from Stack Overflow using StackExchange data explorer [2] for our investigation. Since we are interested in Java APIs, we only collect such questions that are tagged as *java*. Besides, we apply several other constraints–(1) each of the questions should have at least three answers (i.e., average answer count) with one answer being accepted as solution, in order to ensure that the questions are answered substantially and successfully, and (2) the accepted answers should contain code like elements such as code snippets or code tokens so that API information can be extracted from them. We identify the code elements with the help of `<code>` tags in the HTML source of the answers (details in Section II-B), and use Jsoup [5], a popular Java library, for HTML parsing and content extraction.

We collect a total of 2,912 Java classes from 11 core API packages[1] of standard Java edition 6, one of the most stable

[1] https://en.wikipedia.org/wiki/Java_package

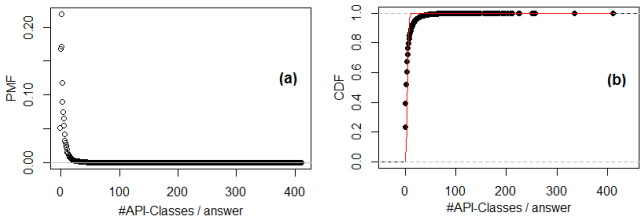


Fig. 2. API frequency distribution– (a) API frequency PMF (b) API frequency CDF

versions [9], for our study. The goal is to find out if these classes are referred to in Stack Overflow posts, and if yes, to what extent they are referred to. We first use Reflections [8], a runtime meta data analysis library, to collect the API classes from JRE 6, and then apply regular expressions on their fully qualified names for extracting the class name tokens. Table I shows class statistics of the 11 API packages selected for our investigation.

We also collect a set of 18,662 real life search queries (of the first author) from Google for over the last eight years which are analyzed to answer the third research question. Although the queries come from a single user, they contain a large vocabulary of 9,029 distinct natural language keywords, and the vocabulary is built over a long period of time. Thus, a study using those queries can produce significant intuitions.

### B. API Class Name Extraction

Several existing studies [11, 15, 25] extract code elements such as API packages, classes and methods from unstructured natural language texts (e.g., forum posts, mailing lists) using information retrieval (e.g., TF-IDF) and island parsing techniques. In the case of island parsing, they apply a set of regular expressions describing Java language specifications [16], and isolate the land (i.e., code elements) from water (i.e., free-form texts). We borrow their parsing technique [25], and apply it to the extraction of API elements from Stack Overflow posts. Since we are interested in the API classes referred to in the posts, we adopt a selective approach for identifying them. We first isolate the code like sections from HTML source of each of the answers from Stack Overflow using `<code>` tags. Then we split the sections based on white spaces and punctuation marks, and collect the tokens having the camel-case notation for Java class (e.g., `HashSet`). According to the existing studies [15, 25], such parsing of code elements sometimes might introduce false positives. Thus, we restrict our exploratory analysis to a closed set of 2,912 API classes from 11 core Java packages (Table I) for avoiding false positives (e.g., camel-case tokens but not valid API classes).

### C. Answering $RQ_1$: API use in accepted SO answers

Since our API recommendation technique exploits keyword-API associations from Stack Overflow, we investigate if the accepted answers actually use certain APIs of our interest in the first place. According to our investigation, out of 172,043 accepted answers, 136,796 (79.51%) answers refer to one or more Java classes (i.e., standard API or user defined), and

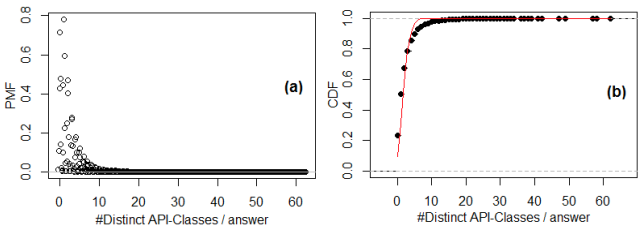


Fig. 3. Distinct API frequency distribution– (a) Distinct API frequency PMF (b) Distinct API frequency CDF

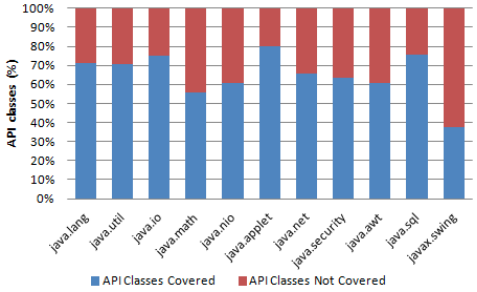


Fig. 4. Coverage of API classes from core packages by Stack Overflow answers

104,983 (61.02%) of them use API classes from the 11 core Java packages (Table I) as a part of their solution.

Fig. 2 shows (a) probability mass function (PMF) and (b) cumulative density function (CDF) for the total frequency of API classes used in each of the Stack Overflow answers where the classes belong to the core Java packages. Both density curves suggest that the frequency observations derive from a heavy-tailed distribution, and majority of the densities accumulate over a short frequency range. The empirical CDF curve also closely matches with the theoretical CDF [1] (i.e., red curve in Fig. 2-(b)) of a Poisson distribution. Thus, we believe that the observations are probably taken from a Poisson distribution. We get a 95% confidence interval over [5.08, 5.58] for mean frequency ($\lambda = 5.32$) which suggests that the API classes from the core packages are referred to at least five times on average in each of the answers from Stack Overflow. We also get $10^{th}$ quantile at frequency=2 and $97.5^{th}$ quantile at frequency=10 which suggest that only 10% of the frequencies are below 3 and only 2.5% of the frequencies are above 10. Fig. 3 shows similar density curves for the frequency of distinct API classes in each of the accepted answers. We get a 95% confidence interval over [2.33, 2.41] for mean frequency ($\lambda = 2.37$) which suggests that at least two distinct classes are used on average in each answer. $30^{th}$ quantile at frequency = 1 and $80^{th}$ quantile at frequency = 4 suggest that 30% of the Stack Overflow answers refer to at least one API class whereas 20% of the answers refer to at least four distinct API classes from the core Java packages.

Thus, to answer **$RQ_1$**, at least two API classes from the core packages are referred to in each of the accepted answers from Stack Overflow that contain API classes from those packages, and they are used at least five times on average in each answer.

### D. Answering $RQ_2$: API coverage in accepted answers

Since our technique exploits mapping between API classes in Stack Overflow answers and keywords from corresponding

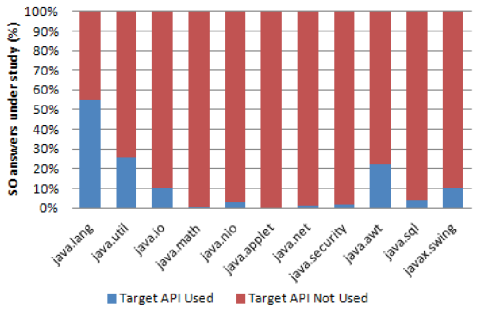


Fig. 5. Use of core packages in Stack Overflow answers

questions for API recommendation, we need to investigate if such answers actually use a significant portion of the API classes from the core packages. We thus identify the occurrence of the classes from core packages in Stack Overflow answers, and determine API coverage for those packages.

Fig. 4 shows the fraction of the classes that are used in Stack Overflow answers for each of the 11 core packages under study. We note that at least 60% of the classes are used in Stack Overflow for nine out of 11 packages. The remaining two packages–`java.math` and `javax.swing` have 55.56% and 37.41% class coverage respectively. Among those nine packages, three large packages– `java.lang`, `java.util` and `java.io` even have a class coverage over 70%. Fig. 5 shows the fraction of Stack Overflow answers (under study) that use API classes from each of the core 11 packages. We note that classes from `java.lang` package are used in over 50% of the answers, which is quite expected since the package contains the frequently used and basic classes such as `String`, `Integer`, `Method`, `Exception` and so on. Two packages– `java.util` and `java.awt` that focus on utility functions (e.g., unzip, pattern matching) and user interface controls (e.g., radio button, check box) respectively have a post coverage over 20%. We also note that classes from `java.io` and `javax.swing` packages are used in over 10% of the Stack Overflow answers, whereas such statistic for the remaining six packages is less than 10%.

Thus, to answer **RQ$_2$**, on average, about 65.15% of the API classes from each of the core Java packages are used in Stack Overflow answers, and at least 12.22% of the answers refer to the classes from each single API package as a part of their solutions. These findings suggest a high potential of Stack Overflow for API recommendation.

### *E. Answering RQ$_3$: Search keywords in SO questions*

Our technique relies on the mapping between natural language tokens from Stack Overflow questions and API classes from corresponding accepted answers for translating a code search query into several relevant API names. Thus, we need to investigate if the texts from such questions actually contain keywords used for code search or not. We are particularly interested in the title of a Stack Overflow question since it summarizes the technical requirement of the question using a few words, and also quite resembles a search query. We analyze the titles of 172,043 Stack Overflow questions and 18,662 real life queries used for Google search. Since we are

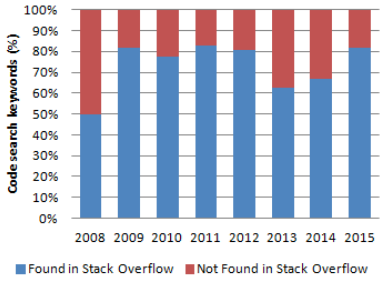


Fig. 6. Coverage of keywords from the collected queries in Stack Overflow questions

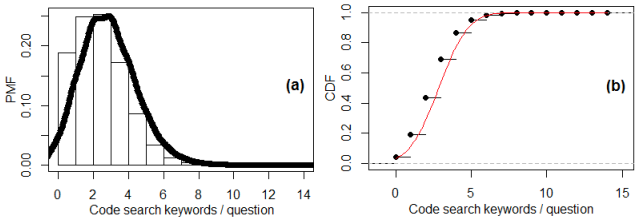


Fig. 7. Collected search query keywords in Stack Overflow– (a) Keyword frequency PMF (b) Keyword frequency CDF

interested in code search queries, we only select those queries that contain any of these keywords–*java, code, example* and *programmatically* for our analysis. A search using such keywords in the query is generally intended for code example search. We get 1,703 such queries containing 1,461 distinct natural language tokens from our query collection.

According to our analysis, the question titles contain 20,391 unique tokens after performing natural language processing (i.e., stop word removal, splitting and stemming), and the tokens match 66.94% of the keywords collected from our code search queries. Fig. 6 shows the fraction of the search keywords that match with the tokens from Stack Overflow questions for the past eight years starting from 2008. We note that on average, 73.03% of the code search keywords from each year match with Stack Overflow tokens. Such statistic reaches up to 80% for the year 2009 to year 2011. One possible explanation for this is that the user (i.e., first author) was a professional developer then, and most of the queries were programming or code example related. Fig. 7 shows (a) probability mass function, and (b) cumulative density function for keyword frequency in the question titles. We note that the density curve shows central tendency like a normal curve (i.e., bell shaped curve), and the empirical CDF also closely matches with the theoretical CDF (i.e., red curve) of a normal distribution with $\mu = 2.85$ and $\sigma = 1.54$. Thus, we believe that the frequency observations come from a normal distribution. We get a mean frequency, $\mu = 2.85$ with 95% confidence interval over [2.84, 2.86], which suggests that each of the question titles from Stack Overflow contains approximately three code search keywords on average.

Thus, to answer **RQ$_3$**, titles from Stack Overflow questions contain a significant amount of the keywords that were used for real life code search. Each title contains approximately three query keywords on average, and their tokens match with about 73% of our collected code search keywords when considered on a yearly basis.

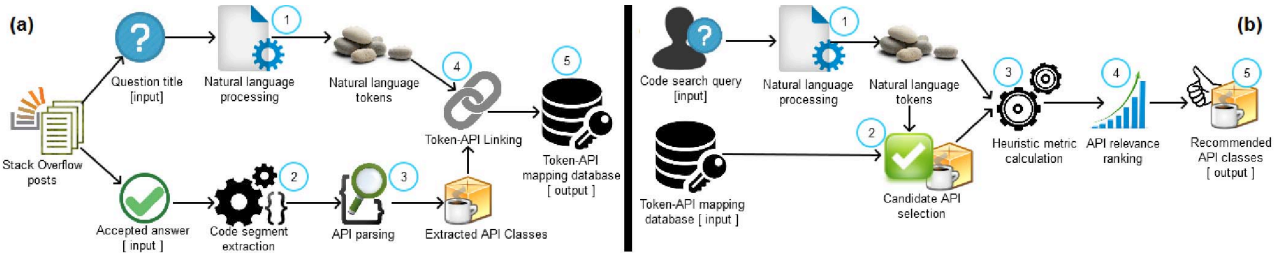


Fig. 8. Proposed technique for API recommendation–(a) Construction of token-API mapping database, (b) Translation of a code search query into relevant API classes

## III. RACK: Automatic API Recommendation using Crowdsourced Knowledge

According to the exploratory study (Section II), at least two API classes are used in each of the accepted answers of Stack Overflow, and about 65% of the API classes from the core packages are used in those answers. Besides, the titles from Stack Overflow questions are a major source for code search keywords. Such findings suggest that Stack Overflow might be a potential source for code search keywords and API classes relevant to them. Since we are interested in exploiting this keyword-API association from Stack Overflow questions and answers for API recommendation, we need a technique that stores such associations, mines them automatically, and then recommends the most relevant APIs. Thus, our proposed technique has two major steps – (a) Construction of token-API mapping database, and (b) Recommendation of relevant APIs for a code search query. Fig. 8 shows the schematic diagram of our proposed technique–RACK– for API recommendation.

### A. Construction of Token-API Mapping Database

Since our technique relies on keyword-API associations from Stack Overflow, we need to extract and store such associations for quick access. In Stack Overflow, each question describes a technical requirement such as *"how to send an email in Java?"* The corresponding answer offers a solution containing code example(s) that refer(s) to one or more API classes (e.g., `MimeMessage`, `Transport`). We capture such requirement and API classes carefully, and exploit their semantic association for the development of token-API mapping database. Since the title summarizes a question using a few words, we only use the titles from the questions. Besides, acceptance of an answer by the person who posted the question indicates that the answer actually meets the requirement in the question. Thus, we consider only the accepted answers from the answer collection for our analysis. The construction of the mapping database has several steps as follows:

**Token Extraction from Titles:** We collect title(s) from each of the questions, and apply standard natural language pre-processing steps such as stop word removal, splitting and stemming on them (Step 1, Fig. 8-(a)). Stop words are the frequently used words (e.g., the, and, some) that carry very little semantic for a sentence. We use a stop word list [10] hosted by Google for the stop word removal step. The splitting step splits each word containing any punctuation mark (e.g., ?,!,;), and transforms it into a list of words. Finally, the stemming step extracts the root of each of the words (e.g., "send" from "sending") from the list, where Snowball stemmer [23, 30] is used. Thus, we extract a set of unique and stemmed words that collectively convey the semantic of the question title, and we consider them as the tokens from the title.

**API Class Extraction:** We collect the accepted answer for each of the questions, and parse their HTML content using Jsoup parser [5] for code segments (Step 2, 3, Fig. 8-(a)). We extract all `<code>` tags from the content as they generally contain code segments [24]. It should be noted that code segments may sometimes be demarcated by other tags or no tag at all. However, identification of such code segments is challenging and often prone to false-positives. Thus, we restrict our analysis to contents inside `<code>` tags for code segment collection from Stack Overflow. We split each of the segments based on punctuation marks and white spaces, and discard the programming keywords. Existing studies [11, 25] apply island parsing for API method or class extraction where they use a set of regular expressions. Similarly, we use a regular expression for Java class [16], and extract the API class tokens having camel case notation. Thus, we collect a set of unique API classes from each of the accepted answers.

**Token-API Linking:** Natural language tokens from a question title hint about the technical requirement described in the question, and API names from the accepted answer represent the relevant APIs that can meet such requirement. Thus, the programming Q & A site–Stack Overflow– inherently provides an important semantic association between a list of tokens and a list of APIs. For instance, our technique generates a list of natural language tokens–{*generat, md5, hash*}– and an API token– `MessageDigest`– from the showcase example on MD5 hash (Fig. 1). We capture such associations from 136,796 Stack Overflow question and accepted answer pairs, and store them in a relational database (Step 4, 5, Fig. 8-(a)) for relevant API recommendation for any code search query.

### B. API Relevance Ranking & Recommendation

In the token-API mapping database, each token associates with different APIs, and each API associates with a number of tokens. Thus, we need a technique that carefully analyzes such associations, identifies the candidate APIs, and then recommends the most relevant ones from them for a given query. It should be noted that we do not apply the traditional association rule mining since our investigations suggest that many token and API sets extracted from our constructed database (Section III-A) have low support. Thus, the mined rules might not be sufficient for API recommendation. The API

ranking and recommendation involve several steps as follows:

*1) Identification of Keyword Context:* In natural language processing, the context of a word refers to the list of other words that co-occur with that word in the same phrase, sentence or even the same paragraph [18]. Co-occurring words complement the semantics of one another [22]. Yuan et al. [34] analyze programming posts and tags from Stack Overflow Q & A site, and use word context for determining semantic similarity between any two software-specific words. In this research, we identify words that co-occur with each keyword in the thousands of question titles from Stack Overflow. For each keyword, we refer to these co-occurring words as its *context*. We then opportunistically use these context words for estimating semantic relevance between any two keywords.

*2) Candidate API Selection:* In order to collect candidate APIs for a query, we employ two heuristics. These heuristics consider not only the association between keywords and APIs but also the coherence among the keywords. Thus, the key idea is to identify such APIs as candidates that are both likely for the query keywords and functionally consistent to one another.

**Keyword-API Co-occurrence (KAC):** Stack Overflow discusses thousands of programming problems, and the discussions contain various natural language keywords and reference to a number of APIs. According to our observation, several keywords might co-occur with a particular API or a particular keyword might co-occur with several APIs across different programming solutions. This co-occurrence generally takes place either by chance or due to semantic relevance. Thus, if carefully analyzed, such co-occurrences could be a potential source for semantic association between keywords and APIs. We capture these co-occurrences (i.e., associations) between keywords and APIs, discard the random associations using a heuristic threshold ($\delta$), and then collect the top APIs ($L[K_i]$) for each keyword ($K_i$) that co-occurred most frequently with the keyword at Stack Overflow.

$$L[K_i] = \{A_j \mid A_j \epsilon A \ \wedge \ rank_{freq}(K_i \rightarrow A_j) \leq \delta\}$$

Here, $K_i \rightarrow A_j$ denotes the association between a keyword $K_i$ and an API $A_j$, $rank_{freq}$ returns rank of the association from the ranked list based on association frequency, and $\delta$ is a heuristic rank threshold. In our research, we consider top five (i.e., $\delta = 5$) APIs for each keyword which is chosen based on iterative experiments on our dataset.

**Keyword-Keyword Coherence (KKC):** Since a code search query might contain multiple keywords (i.e., describing a single task), the candidate APIs should be not only relevant to multiple keywords but also consistent with one another. Yuan et al. [34] determine semantic similarity between any two software specific words by using their contexts from Stack Overflow questions and answers. We adapt their technique for identifying coherent keyword pairs which are then used for collecting candidate APIs functionally relevant to those pairs. We (1) develop a context ($C_i$) for each of the $n$ query keywords by collecting its co-occurred words from thousands of question titles from Stack Overflow, (2) determine semantic similarity for each of the ${}^nC_2$ keyword pairs based on their context derived from Stack Overflow, and (3) use those measures to identify the coherent pairs and then to collect the functionally relevant APIs for them. At the end of this step, we have a set of APIs for each pair of coherent keywords.

Suppose, two query keywords $K_i$ and $K_j$ have context word list $C_i$ and $C_j$ respectively. Now, the candidate APIs ($L_{coh}$) that are relevant to both keywords and functionally consistent with one another can be selected as follows:

$$L_{coh}[K_i, K_j] = \{L[K_i] \cap L[K_j] \mid cos(C_i, C_j) > \gamma\}$$

Here, $cos(C_i, C_j)$ denotes the *cosine similarity* [24] between two context lists, and $\gamma$ is the similarity threshold. We consider $\gamma = 0$ in this work based on iterative experiments on our dataset. $L[K_i]$ and $L[K_j]$ are top frequent APIs for the two keywords– $K_i$ and $K_j$. Thus, $L[K_i, K_j]$ contains such APIs that are relevant to both keywords (i.e., co-occurred with them at Stack Overflow answers), and functionally consistent with one another given that the target keywords are coherent themselves (i.e., semantically similar).

**Algorithm 1** API Relevance Ranking Algorithm

```
1:  procedure RACK(Q)                          ▷ Q: code search query
2:      R ← {}                                 ▷ list of relevant APIs
3:      ▷ collecting keywords from the search query
4:      K ← collectKeywords(Q)
5:      ▷ collecting candidate APIs
6:      L ← getKACList(K)
7:      L_coh ← getKKCList(K)
8:      ▷ estimating relevance of the candidate APIs
9:      for Keyword K_i ∈ K do
10:         sortedAPIs ← sortByFreq(L[K_i])
11:         for APIClass A_j ∈ sortedAPIs do
12:             ▷ likelihood score of an API
13:             S_KAC ← getKACScore(A_j, sortedAPIs)
14:             R[A_j].score ← R[A_j].score + S_KAC
15:         end for
16:     end for
17:     for Keyword K_i, K_j ∈ K do
18:         C_i ← getContextList(K_i)
19:         C_j ← getContextList(K_j)
20:         ▷ relevance of an API with multiple keywords
21:         S_KKC ← getKKCScore(C_i, C_j)
22:         for APIClass A_j ∈ L_coh[K_i, K_j] do
23:             R[A_j].score ← R[A_j].score + S_KKC
24:         end for
25:     end for
26:     ▷ ranking of the APIs
27:     rankedAPIs ← sortByScore(R)
28:     return rankedAPIs
29: end procedure
```

*3) API Relevance Ranking Algorithm:* Fig. 8-(b) shows the schematic diagram, and Algorithm 1 shows the pseudo code of our API relevance ranking algorithm. Once a search query is submitted, we (1) perform Parts-of-Speech (POS) tagging on the query for extracting the meaningful terms such as nouns and verbs [32], and (2) apply standard natural

TABLE II
AN EXAMPLE OF API RECOMMENDATION USING RACK

| java | Scores | | parser | Scores | | html | Scores | | Recommended APIs | Total Score |
|---|---|---|---|---|---|---|---|---|---|---|
| | $S_{KAC}$ | $S_{KKC}$ | | $S_{KAC}$ | $S_{KKC}$ | | $S_{KAC}$ | $S_{KKC}$ | | |
| `List` | 1.00 | 0.20 | `Document` | 1.00 | 0.42 | `Document` | 1.00 | | `Document` | 2.42 |
| `ArrayList` | 0.80 | | `List` | 0.80 | | `Jsoup` | 0.80 | | `File` | 2.10 |
| `File` | 0.60 | 0.20 | `Element` | 0.60 | 0.42 | `Element` | 0.60 | | `List` | 2.00 |
| `Map` | 0.40 | | `File` | 0.40 | 0.42 | `Elements` | 0.40 | | `Element` | 1.62 |
| `Runnable` | 0.20 | | `Node` | 0.20 | | `File` | 0.20 | 0.28 | `Jsoup` | 0.80 |

language processing (i.e., stop word removal, splitting, and stemming) on them to extract the stemmed words (Line 4, Algorithm 1). For example, the query–*"html parser in Java"* turns into three keywords–*'html', 'parser'* and *'java'* at the end of the above step. We then apply the two heuristics–*KAC* and *KKC*– on those stemmed keywords, and collect candidate APIs from the token-API linking database (Step 2, Line 5–Line 7). The candidate APIs are selected based on not only their co-occurrence with the query keywords but also the coherence among the keywords. We then use the following two metrics (derived from the above heuristics) to estimate the relevance of the candidate APIs for the query.

**API Likelihood** estimates the probability of co-occurrence of a candidate API ($A_j$) with an associated keyword ($K_i$) from the search query. It considers the rank of the API in the ranked list based on keyword-API co-occurrence frequency (i.e., $KAC$), and then provides a normalized score as follows.

$$S_{KAC}(A_j, K_i) = 1 - \frac{rank(A_j, sortByFreq(L[K_i]))}{|L[K_i]|}$$

Here, $S_{KAC}$ denotes the API Likelihood estimate, and it ranges from 0 (i.e., not likely at all for the keyword) to 1 (i.e., very much likely for the keyword).

**API Coherence** estimates the relevance of a candidate API ($A_j$) to multiple keywords from the query simultaneously. Since the search query targets a particular programming task, each of the recommended APIs should be relevant to multiple keywords from the query. One way to heuristically determine such relevance is to exploit the semantic similarity between the corresponding keywords that co-occurred with that API. We thus determine semantic similarity between any two keywords ($K_i, K_j$) using their context lists ($C_i, C_j$) [34], and then propagate that measure to each of the candidate APIs ($A_j$) that co-occurred with both keywords (i.e., $KKC$) as a proxy to relevance between the candidate and the two keywords.

$$S_{KKC}(A_j, K_i, K_j) = cos(C_i, C_j) \mid (K_i \rightarrow A_j) \wedge (K_j \rightarrow A_j)$$

Here, $S_{KKC}$ denotes the API Coherence estimate, and it ranges from 0 (i.e., not relevant at all with multiple keywords) to 1 (i.e., very much relevant). It should be noted that each candidate, $A_j$, comes from $L[K_i]$ or $L[K_j]$, i.e., the API is already relevant (i.e., frequently co-occurred) to each of $K_i$ and $K_j$ in their corresponding contexts. $S_{KKC}$ investigates how similar those contexts are, and thus heuristically estimates the relevance of the API, $A_j$, to both keywords.

We first compute *API Likelihood* for each of the candidate APIs that suggests the likeliness of the API for each keyword from the query (Line 9–Line 16). Then we determine *API Coherence* for each candidate API that suggests relevance of the API to multiple keywords from the query (Line 17–Line 25). Once both metrics are calculated for each of the entries from $L$ and $L_{coh}$ (Step 3, Fig. 8-(b)), the scores are accumulated for each individual candidate API (Line 14 and Line 23, Algorithm 1). The candidates are then ranked based on their accumulated scores, and top K APIs from the list are returned for recommendation (Line 26–Line 28, Algorithm 1, Step 4, 5, Fig. 8-(b)).

**Example:** Table II shows a working example on how our API recommendation technique–RACK– works. We first collect the top 5 (i.e., $\delta$) candidate APIs for each of the keywords–*'java','parser'* and *'html'*– based on their co-occurrence frequencies with the keywords. Then we calculate the likelihood (i.e., $S_{KAC}$) of each candidate. For example, `Document` has a maximum likelihood of 1.00 among the candidates for both *'parser'* and *'html'*. We then collect coherence (i.e., $S_{KKC}$) of each candidate API based on semantic relevance among the keywords. For example, *'parser'* and *'html'* have a semantic relevance of 0.42 on the scale from 0 to 1, and that score is added to the overlapping candidates–`Document`, `Element` and `File`– between these two keywords. We then accumulate both scores for each candidate, discard the duplicate candidate APIs (i.e., superclass or subclass), and finally get a ranked list. From the list, we see that `Document`, `Element` and `Jsoup` are highly relevant APIs for the query–*"java parser html"*.

## IV. EXPERIMENT

One of the most effective ways to evaluate a technique for API recommendation is to analyze the relevance of the recommended APIs for a target query. We evaluate our technique using 150 code search queries, determine its performance using four metrics, and then compare with two variants of the state-of-the-art technique. We particularly answer four research questions through our experiments as follows:

- **RQ$_4$:** How does the proposed technique perform in recommending relevant APIs for a code search query?
- **RQ$_5$:** How effective are the proposed heuristics–KAC and KKC–in capturing the relevant APIs for a query?
- **RQ$_6$:** Is our selection of keywords from a given query effective in retrieving the relevant APIs?
- **RQ$_7$:** Can RACK outperform the state-of-the-art technique in recommending APIs for any given set of queries?

### A. Experimental Dataset

**Data Collection:** We collect 150 code search queries for our experiment from three Java tutorial sites– KodeJava [6], JavaDB [4] and Java2s [3]. These sites discuss hundreds of programming tasks that involve the usage of different APIs from the standard Java libraries. Each of these task descriptions generally has three parts–(1) a title (i.e., question) for the

task, (2) one or more code snippets, and (3) an associated prose explaining the code. The title (e.g., "How do I decompress a GZip file in Java?") summarizes the programming task in natural language using a few keywords, and it quite resembles a query for code search as well. We thus use such titles as the code search queries for our experiment in this research.

**Gold Set Development:** The prose explaining the code often refers to one or more APIs (e.g., `GZipOutputStream`, `FileOutputStream`) from the code snippet(s) that are found essential for the task. In other words, such APIs can be considered as the most relevant ones for the target task. We collect such APIs from the prose against each of the task titles from our dataset, and develop a gold set–*API-goldset*–for the experiment. Since relevance of the APIs is determined based on working code examples and their associated prose from the publicly available popular tutorial sites, the subjectivity associated with the relevance is minimized [14].

### *B. Performance Metrics*

We choose four performance metrics for evaluation and validation that are widely used by relevant literature [14, 21, 29]. Two of them are related to recommendation systems whereas the rest two come from information retrieval domain.

**Top-K Accuracy**: It refers to the percentage of the search queries for which at least one API is correctly recommended within the Top-K results by a recommendation technique. Top-K Accuracy can be defined as follows:

$$Top\text{–}K\,Accuracy(Q) = \frac{\sum_{q \in Q} isCorrect(q, Top\text{–}K)}{|Q|}\%$$

Here, $isCorrect(q, Top\text{–}K)$ returns a value 1 if there exists at least one API from the *API-gold set* in the Top-K results, and returns 0 otherwise. $Q$ denotes the set of all search queries.

**Mean Reciprocal Rank@K (MRR@K)**: Reciprocal rank@K refers to the multiplicative inverse of the rank of the first relevant API in the Top-K results returned by a technique. Mean Reciprocal Rank@K (MRR@K) averages such measures for all search queries in the dataset.

**Mean Average Precision@K (MAP@K)**: *Precision@K* calculates the precision at the occurrence of every single relevant API in the ranked list. *Average Precision@K (AP@K)* averages the *precision@K* for all relevant APIs in the list for a code search query. *Mean Average Precision@K* is the mean of *average precision@K* for all queries from the dataset.

**Mean Recall@K (MR@K)**: Recall@K refers to the percentage of gold set APIs that are correctly recommended for a code search query in the Top-K results by a technique. Mean Recall@K (MR@K) averages such measures for all queries.

### *C. Evaluation of RACK*

Each of the queries summarizes a programming task that demands the use of one or more APIs from standard Java libraries. Our technique recommends the top 10 relevant APIs for each query which are then compared with the *API-goldset* for evaluation and validation using the above four metrics.

Table III shows the performance details of our technique for Top-3, Top-5 and Top-10 API recommendation. We see

TABLE III
EXPERIMENTAL RESULTS

| Performance Metric | Top-3 | Top-5 | Top-10 |
|---|---|---|---|
| Top-K Accuracy | 49.33% | 62.67% | **78.67%** |
| Mean Reciprocal Rank@K (MRR@K) | 0.17 | 0.17 | 0.17 |
| Mean Average Precision@K | 30.39% | 33.36% | **34.92%** |
| Mean Recall@K (MR@K) | 23.71% | 33.48% | **45.02%** |

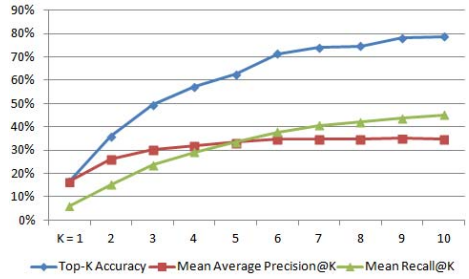


Fig. 9. Top-K Accuracy, Mean Average Precision@K, and Mean Recall@K

TABLE IV
ROLE OF DIFFERENT HEURISTICS

| Heuristics | Metric | Top-3 | Top-5 | Top-10 |
|---|---|---|---|---|
| {Keyword-API Co-occurrence (KAC)} | Accuracy | 50.00% | 66.00% | 78.00% |
| | MRR@K | 0.18 | 0.18 | 0.18 |
| | MAP@K | 31.44% | 34.99% | **35.41%** |
| | MR@K | 23.99% | 34.20% | 44.80% |
| {Keyword-Keyword Coherence (KKC)} | Accuracy | 34.00% | 39.33% | 39.33% |
| | MRR@K | 0.15 | 0.15 | 0.15 |
| | MAP@K | 22.78% | 24.08% | 24.11% |
| | MR@K | 15.24% | 19.02% | 19.52% |
| {Keyword-API Co-occurrence (KAC) & Keyword-Keyword Coherence (KKC)} | Accuracy | 49.33% | 62.67% | **78.67%** |
| | MRR@K | 0.17 | 0.17 | 0.17 |
| | MAP@K | 30.39% | 33.36% | **34.92%** |
| | MR@K | 23.71% | 33.48% | **45.02%** |

that the technique recommends correctly for about 79% of the queries with a mean average precision of 34.92% and a mean recall of 45.02% which are highly promising, especially the Top-K accuracy and the recall, according to relevant literature [14, 21]. While the technique provides a recommendation accuracy of 63.00% for Top-5 results, precision and recall remain close to 33.50%. However, for Top-10 recommendation, the accuracy and recall measures increase significantly, and the precision remains comparable. We do not notice any change in mean reciprocal rank (MRR) for different Top-K recommendations by our technique. Fig. 9 shows how different performance metrics–accuracy, precision and recall change over different values of $K$. We also see that each of these metrics becomes stationary at $K = 10$, which actually supports our choice of $K$ values for top result collection.

Thus, to answer $\mathbf{RQ_4}$, our API recommendation technique–RACK recommends correct APIs for about 79% of the queries with a precision of 34.92% and a recall of 45.02% on average.

We investigate effectiveness of the two applied heuristics–KAC and KKC, and justify their combination in the API ranking algorithm (Algorithm 1). Table IV demonstrates how effective each of the heuristics is in capturing relevant APIs for a given code search query. We see that our technique recommends correctly for 78.00% of the queries with 35.41% precision and 44.88% recall when KAC is considered in isolation. On the other hand, the technique provides at most 40.00% accuracy for Top-10 recommendation with KKC heuristic considered in isolation. However, our technique performs the

TABLE V
EFFECT OF DIFFERENT QUERY TERM SELECTION

| Query Terms | Metric | Top-3 | Top-5 | Top-10 |
|---|---|---|---|---|
| All terms from query | Accuracy | 48.00% | 57.33% | 78.00% |
| | MRR@K | 0.17 | 0.17 | 0.17 |
| | MAP@K | 29.67% | 31.40% | 33.67% |
| | MR@K | 22.71% | 30.29% | 43.67% |
| Noun terms only | Accuracy | 50.00% | 65.33% | 72.67% |
| | MRR@K | 0.22 | 0.22 | 0.22 |
| | MAP@K | 33.17% | 36.56% | **36.79%** |
| | MR@K | 24.71% | 34.33% | 41.60% |
| Verb terms only | Accuracy | 18.67% | 23.33% | 26.67% |
| | MRR@K | 0.07 | 0.07 | 0.07 |
| | MAP@K | 11.44% | 12.71% | 13.23% |
| | MR@K | 7.94% | 11.11% | 12.61% |
| Noun and Verb terms combined | Accuracy | 49.33% | 62.67% | **78.67%** |
| | MRR@K | 0.17 | 0.17 | 0.17 |
| | MAP@K | 30.39% | 33.36% | **34.92%** |
| | MR@K | 23.71% | 33.48% | **45.02%** |

TABLE VI
COMPARISON WITH EXISTING TECHNIQUES

| Technique | Metric | Top-3 | Top-5 | Top-10 |
|---|---|---|---|---|
| Thung et al. [29]-I | Accuracy | 30.00% | 38.67% | 42.00% |
| | MRR@K | 0.19 | 0.19 | 0.19 |
| | MAP@K | 23.33% | 24.62% | 23.53% |
| | MR@K | 13.50% | 18.94% | 25.89% |
| Thung et al. [29]-II | Accuracy | 30.67% | 37.33% | **48.67%** |
| | MRR@K | 0.17 | 0.17 | 0.17 |
| | MAP@K | 23.00% | 23.77% | 23.47% |
| | MR@K | 14.78% | 21.06% | **33.44%** |
| RACK (Proposed technique) | Accuracy | 49.33% | 62.67% | **78.67%** |
| | MRR@K | 0.17 | 0.17 | 0.17 |
| | MAP@K | 30.39% | 33.36% | **34.92%** |
| | MR@K | 23.71% | 33.48% | **45.02%** |

best when both heuristics are used in combination. It provides a maximum of about 79.00% recommendation accuracy with 34.92% precision and 45.02% recall for Top-10 results.

Thus, to answer $\mathbf{RQ_5}$, KAC is found more effective than KKC in capturing relevant APIs. However, combination of both heuristics provides the maximum performance. Thus, their combination for API ranking might be justified.

Since our technique identifies relevant APIs based on their co-occurrence with query keywords, the keywords from each query should be chosen carefully. We extract noun and verb terms from the query (Section III-B), and use them for our experiment. In this section, we investigate if the selection of such terms from the query is effective or not. Table V summarizes our comparative analyses using different set of queries. We see that our technique does not perform well especially for Top-3 and Top-5 results when all terms from a search query are used. The performance improves when only *noun* terms are considered from the query. However, the accuracy and the recall for Top-10 results do not reach the maximum. The performance is also not much interesting when only *verb* terms are considered. However, our technique performs the best especially in terms of accuracy and recall when both the *noun* and the *verb* terms are used together.

Thus, to answer $\mathbf{RQ_6}$, important keywords from a query mainly consist of its noun and verb terms, and our query term selection is found quite effective in retrieving relevant APIs.

### D. Comparison with Existing Techniques

Thung et al. [29] take in a feature request and return a list of relevant API methods both by mining of feature request history and by analysis of textual similarity between the request and the API documentations of those methods. To the best of our knowledge, this is the latest closest study to our work, and thus, we select it for comparison. Since feature request history is not available in our experimental settings, we implement *Description-Based Recommender* module from the technique. We collect API documentations of 3,300 classes from the Java standard libraries (i.e., JDK 6), and develop Vector Space Model (VSM) for each of the classes. In fact, we develop two models for each API class using (1) class header comments only, and (2) both class header and method header comments, and implement two variants– Thung et al.-I and Thung et al.-II for our experiment. We use *Apache Lucene* [7] for VSM development and for textual similarity matching between the API classes and each of the queries from our dataset.

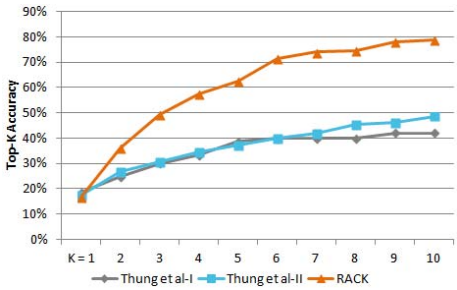

Fig. 10. Top-K Accuracy comparison with existing techniques

Table VI summarizes the comparative analysis between our technique–RACK– and the two variants of Thung et al.. We see that the variants can provide a maximum of about 49% accuracy with 23.47% precision and 33.44% recall for Top-10 results. On the other hand, RACK provides a maximum accuracy of 79% with 34.92% precision and 45.02% recall which are significantly higher. We investigate how the Top-K accuracy changes over different K values for each of these three techniques. From Fig. 10, we see that accuracy for RACK increases gradually up to 79% whereas such performance measures for the textual similarity based techniques stop at 49%. From Fig. 11, we see that RACK performs significantly better than both variants in terms of all three metrics– accuracy, precision and recall. Our median accuracy is above 70% whereas such measure for those variants is below 40%. The same goes for precision and recall measures. Thus, all the findings above suggest that textual similarity between query and API signature or documentations might not be always effective for API recommendation. Our technique overcomes that issue through applying two co-occurrence based heuristics–KAC and KKC– which analyzes the crowdsourced knowledge from Stack Overflow. Performance reported for Thung et al. is project-specific, and the technique is restricted to feature requests [29]. On the contrary, our technique is generic and adaptable for any type of code search. It is also independent of any subject systems. More importantly, it exploits the expertise of a large crowd of technical users for API recommendation which was not considered by the past studies. Thus, our technique possibly has a greater potential.

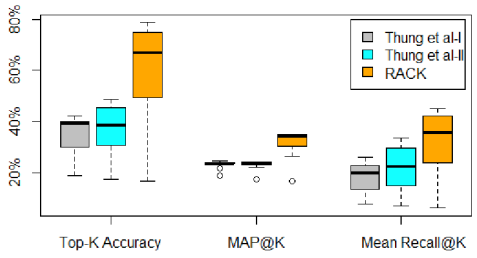


Fig. 11. Comparison with existing techniques

Thus, in order to answer **RQ$_7$**, our proposed technique–RACK– outperforms two variants of the state-of-the-art technique for API recommendation in terms of Top-K accuracy, precision and recall by a large margin.

## V. Threats to Validity

Threats to internal validity relate to experimental errors and biases [34]. We develop *API gold set* for each query by analyzing the code examples and the discussions from tutorial sites which might involve some subjectivity. However, each of the examples is a working solution to the corresponding task (i.e., query), and they are frequently consulted. Thus, the gold set development from sample code is probably a more objective evaluation approach than human judgements of API relevance that introduces more subjective bias [14]. According to the exploratory findings (Section II-D), our technique might be effective only for the recommendation of popular and frequently used APIs. Since fully qualified names are mostly missing in Stack Overflow texts, third-party APIs similar to Java API classes could also have been mistakenly considered.

Threats to external validity relate to the generalizablity of a technique. So far, we experimented using API classes from only standard Java libraries. However, since our technique mainly exploits co-occurrence between keywords and APIs, the technique can be easily adapted for API recommendation in other programming domains.

Threats to construct validity relate to suitability of evaluation metrics. We use Top-K Accuracy and Reciprocal Rank which are widely used for evaluating recommendation systems [26, 29]. The remaining two metrics are well known in information retrieval, and are also frequently used by studies [14, 21, 29] relevant to our work. This confirms no or little threats to construct validity.

## VI. Related Work

**API Recommendation:** Existing studies on API recommendation accept one or more natural language queries, and recommend relevant API classes and methods by analyzing code surfing behaviour of the developers and API invocation chains [21], API dependency graphs [14], feature request history or API documentations [29], and library usage patterns [28]. McMillan et al. [21] first propose *Portfolio* that recommends relevant API methods for a code search query by employing natural language processing, indexing and graph-based algorithms (e.g., PageRank). Chan et al. [14] improve upon *Portfolio*, and return a connected subgraph containing the most relevant APIs by employing further sophisticated graph-mining and textual similarity techniques. Thung et al. [29] recommend relevant API methods to assist the implementation of an incoming feature request by analyzing request history and textual similarity between API details and the request text. In short, each of these relevant techniques above considers lexical similarity between a query and the signature or documentation of the API for collecting candidate APIs, which might not be always effective given that query formulation could be highly subjective. On the other hand, we exploit two co-occurrence heuristics that are derived from crowdsourced knowledge for collecting the candidate APIs, which are found to be relatively more effective. Co-occurrence heuristics overcome the vocabulary mismatch problem [17], and provide a generic, both language and project independent solution. Besides, we exploit the expertise of a large crowd of technical users from Stack Overflow for API recommendation which none of the past studies did. We compare with two two variants of the state-of-the-art technique–Thung et al., and readers are referred to Section IV-D for detailed comparison. Since Thung et al. outperform Chan et al. as reported [29], we compared only with Thung et al. for our validation.

**API Usage Pattern Recommendation:** Thummalapenta and Xie [27] propose *ParseWeb* that takes in a *source object type* and a *destination object type*, and returns a sequence of method invocations that serve as a solution which yields the destination object from the source object. Xie and Pei [33] take a query that describes the method or class of an API, and recommends a frequent sequence of method invocations for the API by analyzing hundreds of open source projects. Warr and Robillard [31] recommend a set of API methods that are relevant to a target method by analyzing the structural dependencies between the two sets. Each of these techniques is relevant to our work since they recommend API methods. However, they operate on structured queries rather than natural language queries, and thus comparing ours with them is not feasible. Of course, we introduced two heuristics and exploited crowd knowledge for API recommendation which were not considered by any of the existing techniques. This makes our contribution significantly different from all of them.

## VII. Conclusion & Future Work

To summarize, we propose a novel technique–RACK– that translates a natural language code search query into a ranked list of relevant APIs. It exploits two novel heuristics derived from crowdsourced knowledge for collecting the relevant APIs. Experiments using 150 code search queries from three Java tutorial sites show that RACK recommends APIs with about 79% Top-10 accuracy which is highly promising. Comparison with two variants of the state-of-the-art technique shows that our technique outperforms both of them in accuracy, precision and recall by a large margin. While that technique is project-sensitive, ours is generic, project independent, and it exploits invaluable crowdsourced knowledge. In future, we plan to apply the co-occurrence heuristics in recommending for other software maintenance tasks such as concept location and traceability link recovery.

## REFERENCES


[1] Theoretical CDF. URL http://stats.stackexchange.com/questions/132652.

[2] Stack Exchange Data Explorer. URL http://data.stackexchange.com/stackoverflow.

[3] Java2s: Java Tutorials, . URL http://java2s.com.

[4] JavaDB: Java Code Examples, . URL http://www.javadb.com.

[5] Jsoup: Java HTML Parser. URL http://jsoup.org.

[6] KodeJava: Java Examples. URL http://kodejava.org.

[7] Apache Lucene Core. URL https://lucene.apache.org/core.

[8] Reflections Library. URL https://code.google.com/p/reflections.

[9] Stable Java version. URL http://stackoverflow.com/questions/6223493.

[10] Stopword List. URL https://code.google.com/p/stop-words.

[11] A. Bacchelli, M. Lanza, and R. Robbes. Linking e-Mails and Source Code Artifacts. In *Proc. ICSE*, pages 375–384, 2010.

[12] S. K. Bajracharya and C. V. Lopes. Analyzing and Mining a Code Search Engine Usage Log. *Empirical Softw. Engg.*, 17(4-5):424–466, 2012.

[13] J. Brandt, P.J. Guo, J. Lewenstein, M. Dontcheva, and S.R. Klemmer. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In *Proc. SIGCHI*, pages 1589–1598, 2009.

[14] W. Chan, H. Cheng, and D. Lo. Searching Connected API Subgraph via Text Phrases. In *Proc. FSE*, pages 10:1–10:11, 2012.

[15] B. Dagenais and M.P. Robillard. Recovering Traceability Links between an API and its Learning Resources. In *Proc. ICSE*, pages 47–57, 2012.

[16] J. Gosling, B. Joy, G. Steele, and G. Bracha. The Java Language Specification: Java SE 7 Edition. 2012.

[17] S. Haiduc and A. Marcus. On the Effect of the Query in IR-based Concept Location. In *Proc. ICPC*, pages 234–237, June 2011.

[18] Z. Harris. Mathematical Structures in Language Contents. 1968.

[19] K. Kevic and T. Fritz. Automatic Search Term Identification for Change Tasks. In *Proc. ICSE*, pages 468–471, 2014.

[20] K. Kevic and T. Fritz. A Dictionary to Translate Change Tasks to Source Code. In *Proc. MSR*, pages 320–323, 2014.

[21] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: Finding Relevant Functions and their Usage. In *Proc. ICSE*, pages 111–120, 2011.

[22] R. Mihalcea and P. Tarau. Textrank: Bringing Order into Texts. In *Proc. EMNLP*, pages 404–411, 2004.

[23] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza. Mining StackOverflow to Turn the IDE into a Self-confident Programming Prompter. In *Proc. MSR*, pages 102–111, 2014.

[24] M. M. Rahman, S. Yeasmin, and C. K. Roy. Towards a Context-Aware IDE-Based Meta Search Engine for Recommendation about Programming Errors and Exceptions. In *Proc. CSMR-WCRE*, pages 194–203, 2014.

[25] P.C. Rigby and M.P. Robillard. Discovering Essential Code Elements in Informal Documentation. In *Proc. ICSE*, pages 832–841, 2013.

[26] P. Thongtanunam, R. G. Kula, N. Yoshida, H. Iida, and K. Matsumoto. Who Should Review my Code ? In *Proc. SANER*, pages 141–150, 2015.

[27] S. Thummalapenta and T. Xie. Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web. In *Proc. ASE*, pages 204–213, 2007.

[28] F. Thung, D. Lo, and J. Lawall. Automated Library Recommendation. In *Proc. WCRE*, pages 182–191, 2013.

[29] F. Thung, S. Wang, D. Lo, and J. Lawall. Automatic Recommendation of API Methods from Feature Requests. In *Proc. ASE*, pages 290–300, 2013.

[30] C. Vassallo, S. Panichella, M. Di Penta, and G. Canfora. Codes: Mining Source Code Descriptions from Developers Discussions. In *Proc. ICPC*, pages 106–109, 2014.

[31] F. W. Warr and M. P. Robillard. Suade: Topology-Based Searches for Software Investigation. In *Proc. ICSE*, pages 780–783, 2007.

[32] E. Wong, J. Yang, and L. Tan. AutoComment: Mining Question and Answer sites for Automatic Comment Generation. In *Proc. ASE*, pages 562–567, 2013.

[33] T. Xie and J. Pei. MAPO: Mining Api Usages from Open Source Repositories. In *Proc. MSR*, pages 54–57, 2006.

[34] T. Yuan, D. Lo, and J. Lawall. Automated Construction of a Software-specific Word Similarity Database. In *Proc. CSMR-WCRE*, pages 44–53, 2014.